11-Puzzle

11-Puzzle Problem AI solver for CS-UY 4613

By Kevin Lee (KL3642)

This program uses Weighted A* Search to search for the optimal set of moves to achieve the goal state from the given initial state.

How to use

1. Create project directory that includes `puzzle.py` and `Node.py` (These are the only two core files needed)
2. Open shell or terminal in project dir (`C:\\...\\11-Puzzle` or however you named it).
3. Use the following syntax:
    1. `python3 puzzle.py 'text_file.txt' 'w'`
        1. Note: All files used must be present in project dir including input files.
    2. For example: `python3 puzzle.py Input1.txt 1.2`
4. The program will create an `output.txt` file in the same directory
    1. Note: Repeated usage of the program will continually append to the file `output.txt`
    2. If you wish to run the program multiple times, you must either rename `output.txt` or delete it.

Future Work

- Redesign structure of code base to be more modular and general
    - Instead of coding Manhattan Distance Heuristic into the main code, abstract by creating a Heuristic class with different heuristics inheriting from the main class
    - Create Problem class with the same principles as Heuristic, where different versions of sliding puzzles can be solved (8-Puzzle, 15-Puzzle, etc.)
    - This will require changing various functions of code including input parsing

Source Code

puzzle.py

```
"""
CS-UY 4613
Source Code for Project 1: 11-Puzzle Problem

Author: Kevin Lee (KL3642)

Description: Implement the A* search algorithm
with graph search for solving the 11-puzzle problem.
"""

# Standard Libraries
import sys
import copy
from heapq import heapify, heappush, heappop  # Heap data structure for priority queue

# Custom Libraries
import Node  # Node data structure



# Heuristic
def manhattan(curr_state, goal_state, w):
    mdist = 0

    # Iterate through curr_state
    for i in range(len(curr_state)):
        if curr_state[i] == 0:
            continue
        # Iterate through goal_state to find match
        for j in range(len(goal_state)):

            # When match is found, calculate Manhattan Distance
            if curr_state[i] == goal_state[j]:
                x1 = i // 4
                y1 = i % 4

                x2 = j // 4
```

```
                y2 = j % 4

                # Add to total mdist
                mdist += abs(x2 - x1) + abs(y2 - y1)

                # Break loop to go to next tile
                break

    return mdist * w


# Create and return a list of possible actions from given state
# Used in expand()
def expand_actions(state):
    poss_actions = []

    # Find index of blank
    ind = 0

    for i in range(len(state)):
        if state[i] == 0:
            ind = i

    # Determine possible actions
    if ind % 4 != 0:
        poss_actions.append('L')

    if ind % 4 != 3:
        poss_actions.append('R')

    if ind > 3:
        poss_actions.append('U')

    if ind < 8:
        poss_actions.append('D')

    return poss_actions


# Create new state based on action
# Used in expand()
def result(state, action):
    new_s = copy.deepcopy(state)

    # Find index of blank
    ind = 0

    for i in range(len(state)):
        if state[i] == 0:
            ind = i

    # Perform action
    if action == 'L':
        new_s[ind] = new_s[ind - 1]
        new_s[ind - 1] = 0

    elif action == 'R':
        new_s[ind] = new_s[ind + 1]
        new_s[ind + 1] = 0

    elif action == 'U':
        new_s[ind] = new_s[ind - 4]
        new_s[ind - 4] = 0

    elif action == 'D':
        new_s[ind] = new_s[ind + 4]
```

```
        new_s[ind + 4] = 0

    return new_s


# Returns all possible children of given node
def expand(parent, goal_state, w):
    s = parent.state

    # Create nodes from all possible actions
    for action in expand_actions(s):
        # Determine the attributes of each child
        new_depth = parent.depth + 1
        gn = parent.path_cost + 1
        new_s = result(s, action)
        fn = gn + manhattan(new_s, goal_state, w)

        yield Node.Node(new_depth, gn, fn, new_s, parent, action)


# A* Search
def search(ini_state, goal_state, w):
    # Initialize Root Node
    node = Node.Node(0, 0, manhattan(ini_state, goal_state, w), ini_state)
    total_num_nodes = 1

    # Initialize frontier
    frontier = [node]
    heapify(frontier)

    # Initialize visited Hash Map
    visited = {}

    # Start Search from Frontier
    while len(frontier) > 0:
        next_node = heappop(frontier)

        # If Goal Node is found
        if next_node.state == goal_state:
            return next_node, total_num_nodes

        # Expand children
        for child in expand(next_node, goal_state, w):
            s = child.state

            # Check if already visited
            if tuple(s) not in visited or child.total_cost < visited[tuple(s)].total_cost:
                visited[tuple(s)] = child
                heappush(frontier, child)

                total_num_nodes += 1

    return None, total_num_nodes


# Explore solution path and return actions and f(n) along the path
# Used in output()
def find_path(node):
    sol_path = []
    fn_vals = []

    # Go up solution path while appending action to sol_path and f(n) to fn_vals until root
    curr = node
    while curr.depth != 0:
        sol_path.append(curr.action)
```

```
        fn_vals.append(curr.total_cost)
        curr = curr.parent

    # Include root node f(n)
    fn_vals.append(curr.total_cost)

    sol_path.reverse()
    fn_vals.reverse()
    return sol_path, fn_vals


# Create and write to output file
def output(ini_state, w, goal_node, num_nodes):
    f = open("output.txt", 'a')

    # Write initial state
    for i in range(len(ini_state)):
        f.write(str(ini_state[i]))
        if i % 4 != 3:
            f.write(' ')
        else:
            f.write('\n')

    f.write('\n')

    # Write goal state
    for i in range(len(goal_node.state)):
        f.write(str(goal_node.state[i]))
        if i % 4 != 3:
            f.write(' ')
        else:
            f.write('\n')

    f.write('\n')

    # Write w
    f.write(str(w) + '\n')

    # Write depth of shallowest goal node
    if goal_node is None:
        f.write("FAIL\n")
    else:
        f.write(str(goal_node.depth) + '\n')

    # Write the total number of nodes generated
    f.write(str(num_nodes) + '\n')

    # Write the solution (sequence of actions and f(n) values)
    if goal_node is None:
        f.write("FAIL\nFAIL")
    else:
        sol_path, fn_vals = find_path(goal_node)
        for i in range(len(sol_path)):
            f.write(sol_path[i])
            if i < len(sol_path) - 1:
                f.write(' ')

        f.write('\n')

        for i in range(len(fn_vals)):
            f.write('{0:.1f}'.format(fn_vals[i]))
            if i < len(fn_vals) - 1:
                f.write(' ')

    f.close()
```

```
# Main
def main():
    # Grab filename and w from stdin
    filename = sys.argv[1]
    w = float(sys.argv[2])

    # Open File and parse input
    f = open(filename)
    lines = f.readlines()

    # Grab states
    ini_state = []
    goal_state = []
    count_line = 0

    for line in lines:
        if count_line >= 3:
            goal_state.extend([int(n) for n in line.split(' ') if n != '\n'])
        else:
            ini_state.extend([int(n) for n in line.split(' ') if n != '\n'])
        count_line += 1

    f.close()

    # Start Search
    res, num_nodes = search(ini_state, goal_state, w)

    # Create and write to output
    output(ini_state, w, res, num_nodes)


if __name__ == "__main__":
    main()
```

Node.py

```
"""
CS-UY 4613
Source Code for Project 1: 11-Puzzle Problem

Author: Kevin Lee (KL3642)

Description: This is the Node data structure used in the project.
"""


class Node:
    def __init__(self, depth=0, path_cost=0, total_cost=0, state=None, parent=None, action=None):
        if state is None:
            state = []
        self.total_cost = total_cost
        self.action = action
        self.parent = parent
        self.depth = depth
        self.path_cost = path_cost
        self.state = state

    def __lt__(self, other):
        return self.total_cost < other.total_cost

    def __repr__(self):
        return "depth = % s , path_cost = % s , total_cost = % s , state = % s , action = % s" % (
            self.depth, self.path_cost,
            self.total_cost, self.state, self.action)
```

Output Files

output1a.txt

```
2 0 6 4
3 10 7 9
11 5 8 1

2 10 6 4
11 3 8 9
0 7 5 1

1.0
7
23
D R D L U L D
7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0
```

output1b.txt

```
2 0 6 4
3 10 7 9
11 5 8 1

2 10 6 4
11 3 8 9
0 7 5 1

1.2
7
23
D R D L U L D
8.4 8.2 8.0 7.8 7.6 7.4 7.2 7.0
```

output1c.txt

```
 2 0 6 4
3 10 7 9
11 5 8 1

2 10 6 4
11 3 8 9
0 7 5 1

1.4
7
23
D R D L U L D
9.8 9.4 9.0 8.6 8.2 7.8 7.4 7.0
```

output2a.txt

```
 2 0 6 4
3 10 7 9
11 5 8 1

2 7 8 4
10 6 9 1
3 11 0 5

1.0
13
33
R D D L L U R U R D R D L
13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0
```

output2b.txt

```
 2 0 6 4
3 10 7 9
11 5 8 1

2 7 8 4
10 6 9 1
3 11 0 5

1.2
13
33
R D D L L U R U R D R D L
15.6 15.4 15.2 15.0 14.8 14.6 14.4 14.2 14.0 13.8 13.6 13.4 13.2 13.0
```

output2c.txt

```
 2 0 6 4
3 10 7 9
11 5 8 1

2 7 8 4
10 6 9 1
3 11 0 5

1.4
13
33
R D D L L U R U R D R D L
18.2 17.8 17.4 17.0 16.6 16.2 15.8 15.4 15.0 14.6 14.2 13.8 13.4 13.0
```

output3a.txt

```
 8 7 2 4
10 6 9 1
0 11 5 3

10 6 8 4
9 7 0 2
11 5 3 1

1.0
17
170
R U R U L L D R D R R U L U L D R
13.0 13.0 15.0 15.0 15.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0 17.0
```

output3b.txt

```
 8 7 2 4
10 6 9 1
0 11 5 3

10 6 8 4
9 7 0 2
11 5 3 1

1.2
17
127
R U R U L L D R D R R U L U L D R
15.6 15.4 17.6 17.4 17.2 19.4 19.2 19.0 18.8 18.6 18.4 18.2 18.0 17.8 17.6 17.4 17.2 17.0
```

output3c.txt

```
 8 7 2 4
10 6 9 1
0 11 5 3

10 6 8 4
9 7 0 2
11 5 3 1

1.4
17
127
R U R U L L D R D R R U L U L D R
18.2 17.8 20.2 19.8 19.4 21.8 21.4 21.0 20.6 20.2 19.8 19.4 19.0 18.6 18.2 17.8 17.4 17.0
```